EECS 427
Discussion 6: Verilog HDL
Reading: Many references

Online Verilog Resources

- ASICs the book, Ch. 11:
  - http://www.ge.infn.it/~pratolo/verilog/VerilogTutorial.pdf
- Verilog Quick Reference Guide:
  - http://www.sutherland-hdl.com/on-line_ref_guide/vlog_ref_top.html
- Alternate Verilog FAQ:
  - http://www.angelfire.com/in/verilogfaq/index.html
- Verilog Introduction
  - http://www.see.ed.ac.uk/~gerard/Teach/Verilog/index.html
- Newsgroup:
  - http://groups.google.com/groups?group=comp.lang.verilog

Topic Outline

- Introduction
- Verilog Background
- Connections
- Modules
- Procedures
- Structural
- Behavioral
- Testbenches
- Simulation

Most slides courtesy of Andrew Kahng, UCSD

Lecture Overview

- Learn Verilog basics
  - Hardware Description Language semantics
  - Verilog Syntax
  - Features
- Behavioral vs. structural Verilog
- Synthesizable Verilog (subset of Verilog itself)
- A few small examples
- A Synth + APR tutorial will be made available on the course website

High-level view of Verilog

- Verilog descriptions look like programs:

C/C++	Verilog
Function	Module
Procedure Parameters	Ports
Variables	Wires/Regs

- Modules resemble subroutines in that you can write one description and use (instantiate) it in multiple places
- Block structure is a key principle
  - Use hierarchy/modularity to manage complexity
- But they aren't 'normal' programs
  - Module evaluation is concurrent (every block has its own "program counter")

Introduction - Motivation

- Generic HDL uses:
  - Simulation
    - Test without build
    - ModelSim, NCVerilog, VCS, etc.
  - Synthesis (build)
    - Real hardware (gates)
    - Design Compiler

Quick Verilog History

- Verilog HDL (Hardware Description Language) was developed by Gateway Design Automation in 83-84
- Put in the public domain by Cadence Design Systems in 1990 to promote the language as a standard
  - Became an IEEE standard in 1995

Hardware Description Languages

- Need a description one level up from logic gates
- Work at the level of functional blocks, not logic gates
  - Complexity of the functional blocks is up to the designer
  - A functional unit could be an adder, or even a microprocessor
- The description consists of functional blocks and their interconnections
  - Describe functional block (not predefined)
  - Support hierarchical description (function block nesting)
- To make sure the specification is correct, create a testbench and run it through NCVerilog (or similar)

Slide courtesy of Ken Yang, UCLA

Verilog Naming Conventions

- The following is used in all code:
  - Two slashes "//" are used to begin single line comments
    - However "// synopsys" is a directive to Design Compiler to do something (we'll show the most common example later)
  - A slash and asterisk "/*" are used to begin a multiple line comment and an asterisk and slash "*/" are used to end a multiple line comment.
  - Names can use alphanumeric characters, the underscore "_" character, and the dollar "$" character
  - Names must begin with an alphabetic letter or the underscore.
  - Spaces are not allowed within names
- Parameter naming; use compiler directives
  - `` `define word_size 16 ``
  - Whenever you see `` `word_size `` → this will be interpreted as 16

Reserved Keywords

- The following is a list of the Verilog reserved keywords:

always	endmodule	medium	reg	tranif0
and	endprimitive	module	release	tranif1
assign	endspecify	nand	repeat	tri
attribute	endtable	negedge	rnmos	tri0
begin	endtask	nmos	rpmos	tri1
buf	event	nor	rtran	triand
bufif0	for	not	rtranif0	trior
bufif1	force	notif0	rtranif1	trireg
case	forever	notif1	scalared	unsigned
casex	fork	or	signed	vectored
casez	function	output	small	wait
cmos	highz0	parameter	specify	wand

Reserved Keywords (continued)

deassign	highz1pmos	param	spec	weak0
default	if	posedge	strength	weak1
defparam	ifnone	primitive	strong0	while
disable	initial	pull0	strong1	wire
edge	inout	pull1	supply0	wor
else	input	pulldown	supply1	xnor
end	integer	pullup	table	xor
endattribute	join	remos	task	
endcase	large	real	time	
endfunction	macromodule	realtime	tran	

Numbers

- Number notation:
  - <size> ‘ <base format><number>
- Examples:
  - `4'b1111 // 4 bit binary number`
  - `12'habc //12 bit hexadecimal number`
  - `16'd255 //16 bit decimal number`
- Z is high impedance, X is don’t care, ? = 0 or 1 or X

Operators

- Arithmetic
  - `* multiply`
  - `/ divide`
  - `+ add`
  - `- subtract`
  - `% modulus`
- Logical
  - `! Not`
  - `&& and`
  - `|| or`
- Relational
  - `> greater`
  - `< less`
  - `>= greater-equal`
  - `<= less-equal` (also used for non-blocking assignments, later)
- Equality
  - `== equal`
  - `!= not equal`
  - `===  (case equality)`
- Bitwise
  - `~ negation`
  - `& and`
  - `\ or`
  - `^ xor`
  - `^~ xnor`
- Conditional
  - `? : if then else`
- Bit munging
  - { } Concatenation
  - [ ] Bit slicing

Connections: Ports

- Keywords:
  - `input` - input
  - `output` - output
  - `inout` - bi-directional
- Ports do not store information
- Example

```
module ex (a, b, c, out)
  output out;
  input a, b, c;
endmodule
```

Wires

- Wires
  - Connection between hardware elements (visualize as a node in the circuit)
  - Module connections
  - Used to connect signals from sensitivity list
  - Memoryless
    - Must be continuously driven by an assignment statement (**assign)**
  - Assigned outside of always blocks
- Example:

```
wire a; // declared wire net
wire b = 1'b0 // tied to zero at declaration
```

(alternatively:

```
wire b;
assign b = 1'b0;
```


Memory Elements

- Register
  - Keyword = reg
  - Represents storage in that its value is whatever was most recently (procedurally) assigned to it
    - But it does NOT necessarily instantiate an actual register
  - Assigned within always blocks
- Examples:

```
reg clock; // clock
reg [0:4] vec_reg // 5 bit register vector
```

Modules

- Primary unit in Verilog
  - Functional block (can be big; ex: ALU)
  - Keywords = module/ endmodule
  - Used for all dataflow types

Procedural Statements

- Control statements
  - Keyword `always` provides functionality of a tiny program that executes repeatedly (usually on some trigger condition, more later)
  - Don't assign a value to a specific `reg` in two different always blocks as it will generate 2 FFs and combine outputs
- Inside an `always` block, can use standard control flow statements:
  - `if (<conditional>) then <statements> else <statements>;`
  - `case (<var>) <value>: <statements>; … default: <statements>`
  - Case statements are prioritized
    - The second case entry can't happen unless the first does not match
    - May not be what the actual hardware implies – especially when cases are mutually exclusive
    - Need additional directives (parallel-case, shown later) to indicate this

```
Example:
    always @ (Activation List)
        begin
                        if (x==y) then
                        out= in1
                        else
                        out = in2;
        end
```

`Initial` Block

- A type of procedural block
  - Does not need an activation list
  - It runs just once, when the simulation starts
- Used at the very start of simulation
  - Initialize simulation environment
  - Initialize design
    - This is usually only used in the first pass of writing a design
    - NOT synthesizable, real hardware does not have initial blocks
  - Allows testing of a design (outside of the design module)

Blocking vs Non-blocking

- Relates to scheduling of events
- Blocking
  - Ex:

```
begin
  A = B;
  B = A;
end
```

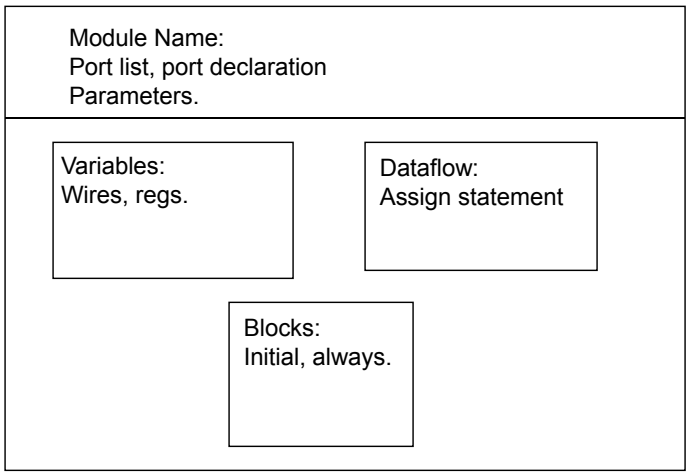
  - Each assignment is completed before moving to the next line
  - In this case, value held in B is assigned to A, and then the value assigned in A (same as in B) is then assigned back to B.
- Non-blocking (preferable in sequential elements)
  - Ex:

```
begin
  A <= B;
  B <= A;
end
```

  - Values on RHS of both expressions are held in temp locations, all assignments are done concurrently → A and B are swapped

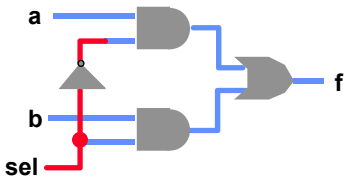
Modules

- Structure



Structural Verilog

- Structural models
  - Are built from gate primitives and/or other modules
  - They describe the circuit using logic gates — much as you would see in an implementation of a circuit
  - Basically you are clearly specifying the structure of the circuit
- Identify:
  - Gate instances, wire names, delay from *a* or *b* to *f*.




```
module mux (f, a, b, sel);
   output f;
   input  a, b, sel;

   and #5 g1 (f1, a, nsel),
          g2 (f2, b, sel);
   or    #5  g3 (f, f1, f2);
   not    g4 (nsel, sel);
endmodule
```

Behavioral Modeling

- More abstract, no direct description of how a module is implemented using primitives
- Mux using behavioral: `assign f = (sel) ? a: b;`
- *Procedural* statements are used
  - Statements using "always" Verilog construct
  - Can specify both combinational and sequential circuits
- Normally don't think of procedural stuff as "logic"
  - They look like C: mix of ifs, case statements, assignments …
  - … but there is a semantic interpretation to put on them to allow them to be used for simulation and synthesis (giving equivalent results)

Slide courtesy of Don Thomas, Carnegie Mellon

Behavioral Statements

- if-then-else
  - What you would expect, except that it's doing 4-valued logic. 1 is interpreted as True; 0, x, and z are interpreted as False

```
if (select == 1)
     f = in1;
else f = in0;
```

- case
  - What you would expect, except for 4-valued logic
  - There is no *break* statement — it is assumed
  - Casex statement treats Z/X as don't cares

```
case (select)
  2'b00: a = b + c;
  2'b01: q = r + s;
  2'bx1: r = 5;
  default: r = 0;
endcase
```


Slide courtesy of Don Thomas, Carnegie Mellon

Activation Lists

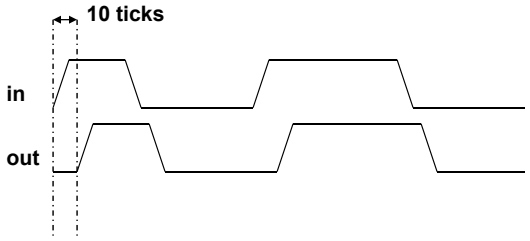
- Contained in `always` block
- Definition: Activation List
  - Tells the simulator when to run this block
    - NOTE!! If not all inputs are sensitized, a latch is created to hold state in those undefined cases
- Activation lists in Verilog:
  - `@(signalName or signalName or …)`
    - Evaluate this block when any of the named signals change (either positive or negative change)
  - `@(posedge signalName);` or `@(negedge signalName);`
    - Makes an edge triggered flip-flop
    - Evaluates only on one edge of a signal
    - Can have `@(posedge signal1 or negedge signal2)`
    - Only allow “or” not “and” because edges are singular events

Testbenches: Delay Models

- Verilog simulation time
  - Execution time of the verilog model
    - When the computer completes with all the “events” that occur at the current simulated time
    - The computer increases time until another signal is scheduled to change values
- Behavioral delay assignments within the blocks
  - `# delayAmount`
    - Simulator sees this symbol, and stops evaluation
    - Pause `delayAmount` of simulated time (# of ticks)
    - Delays are often used to model the delay in functional units
    - Can be tricky to use properly
  - Synthesis does not deal with delays (it computes delays itself)
    - Use only in testbench
    - Synthesizer will ignore

Declarative Delay Control

- A way to specifying delay of a signal
- Make out a delayed version of the input (by 10 ticks)
  - `assign #10 out = in;`
  - Delayed assignment
- Anywhere else to put delay is not allowed
  - `assign out = #10 in;` // is not allowed



Slide courtesy of Ken Yang, UCLA

Building and testing a module

- Construct a "testbench" for your design
  - Develop your hierarchical system within a module that has input and output ports (called "design" here)
  - Develop a separate module to generate tests for the module ("test")
  - Connect these together within another module ("testbench")

```
module testbench ();
    wire   l, m, n;

    design d (l, m, n);
    test   t (l, m);

    initial begin
       //monitor and display
       …
```

```
module design (a, b, c);
    input  a, b;
    output c;
    …
```

```
module test (q, r);
   output    q, r;

   initial begin
     //drive the outputs with signals
       …
```


Slide courtesy of Don Thomas, Carnegie Mellon

Examples: Creating Code

- Example:
  - Given a specification – “build full adder”
  - Name signals:
    - Inputs: carry_in, A, B
    - Outputs: carry_out, sum
  - Math:
    - Sum = (A xor B xor carry_in)
    - Carry_out = (A · B) + carry_in · (A · B)
  - Need:
    - Module name
    - Algorithm (see math)

Full-adder Code

- Sample Code

Sensitivity List

```
module full_adder (a, b, ci, sum, co);  //← lists full
   input/output signal list
input  a, b, ci;       //input declaration
output sum, co;        //output declaration
    assign sum = a ^ b ^ ci;
    assign co = (a & b) | (a & ci) | (b & ci);
endmodule
```

Positive edge-triggered registers with resets

```
module ff1(d,clk,reset,q)
 input d, clk, reset;
 output q;
 reg q;

always @(posedge clk)
  if (reset == 1)
      q <= 0;
  else
      q <= d;
```

OR

```
always @(posedge clk or posedge reset)
  if (reset) q <= 0;
  else
      q <= d;

endmodule
```